Migrating a Flash application to Flex

Author: Aral Balkan

Copyright © 2004 Aral Balkan. All Rights Reserved.
Released under the Creative Commons Attribution-ShareAlike 2.0 England & Wales License.

In this article I will show you how easy it can be to migrate a *well-architected* Macromedia Flash application to Macromedia Flex. In the process, you will see that the two technologies are more similar than different and learn about best-practices, pattern-based methodologies that lower the risk of developing Rich Internet Applications in the Flash ecosystem.

The Flash Ecosystem

Macromedia uses the term "ecosystem" to refer to the family of products that in some way utilize the virtual machine we know as the Flash Player. This includes the Flash IDE development tool, Flex presentation server and Flex Builder development tool. Both Flash and Flex can be scripted using the ECMAScript 4-based ActionScript 2 language and both produce SWF files, the compiled bytecode that is interpreted by the Flash virtual machine. Where they differ greatly is in the source formats used and the time of SWF generation.

The Source of it All

The first big difference between Flash and Flex is that Flash uses a binary FLA file to store its application structure whereas Flex uses a text-based one based on Macromedia's markup language for Flex, MXML. Additionally, both Flash and Flex can be scripted using ActionScript, which itself can either be stored as part of an FLA or MXML file or kept externally in text-based class files.

Not having any of its source code in a binary format gives Flex the advantage of being easier to version control and *diff* (automatically create a line-by-line review of changes between different versions of the same file.) However, as you will see, there are ways to lessen the risk associated with using Flash's proprietary FLA format by using best-practices, pattern-based methodologies in your development process. Such an approach has the added benefit of making it easier to port applications from Flash to Flex.

A Matter of Timing

The second big difference between Flash and Flex is the time of SWF generation.

In Flash, SWF files are generated from the source FLA and ActionScript class files by the Flash IDE at development time. These SWF files can then be deployed to staging and production servers and served like any other file by the web server without need for additional server-side processing.

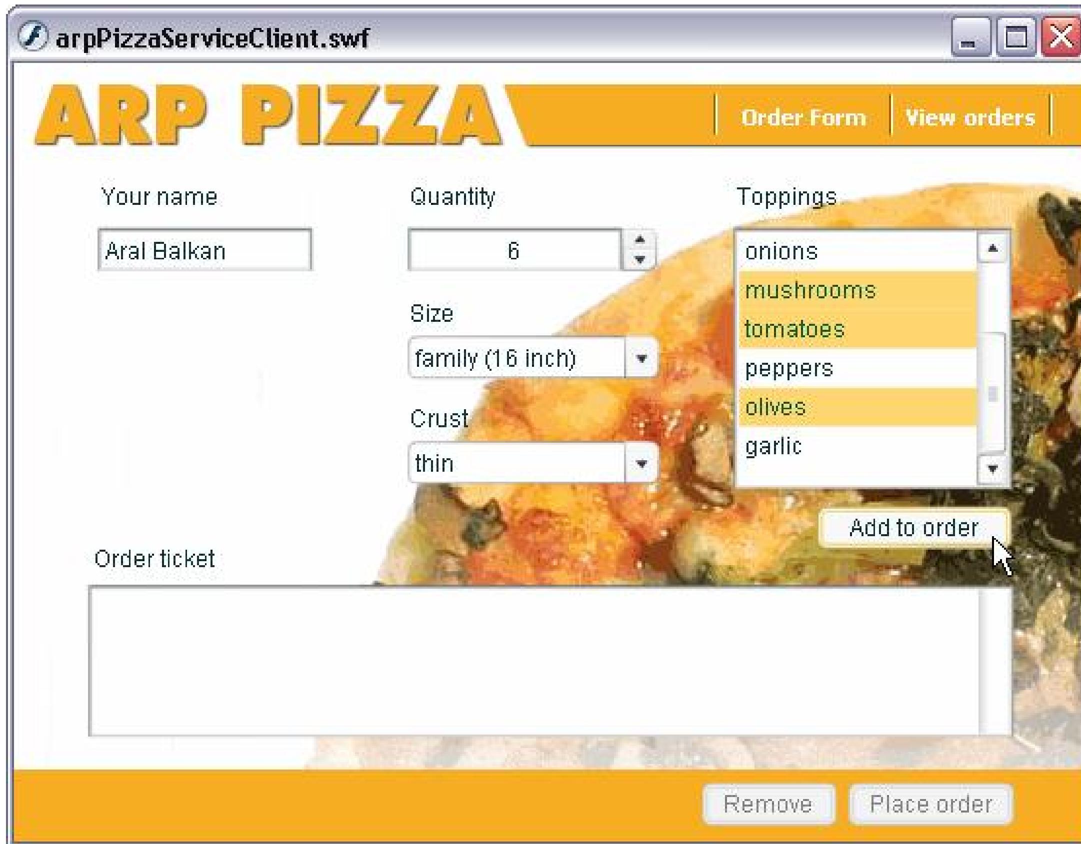
In the Flex development model, developers use either text editors or the Flex Builder IDE to create their MXML source and ActionScript classes. These are then deployed to the Flex server which compiles the MXML and ActionScript files and generates SWF files at runtime. This occurs when the MXML file is first requested and, subsequently, whenever the MXML file is changed. Both approaches have their advantages and disadvantages, a discussion of which is beyond the scope of this article.

Now that you know the two main differences between Flash and Flex and have some perspective on how the two technologies relate to each other, let's take a look at a sample Flash application and explore how its architecture determines the ease with which you can migrate it from Flash to Flex.

ARP Pizza: Flash Version

ARP Pizza is one of the sample applications that comes with the Ariaware RIA Platform (ARP) -- the open-source, best-practices, pattern-based ActionScript 2 framework for Flash and Flex-based RIA development. It is a make-believe online pizza store where users can order pizzas and view existing orders (and all without spending a single dime or actually receiving any pizza, for that matter!).

The front-end is built in Flash and the back-end uses a combination of PHP and MySQL. The communication between the tiers is established via AMFPHP (http://amfphp.org/), an open source PHP implementation of Flash Remoting. In fact, ARP Pizza is a pattern-based remake of the AMFPHP Pizza Service sample application, which you can find on the AMFPHP home page.



[ARP Pizza: Flash version]

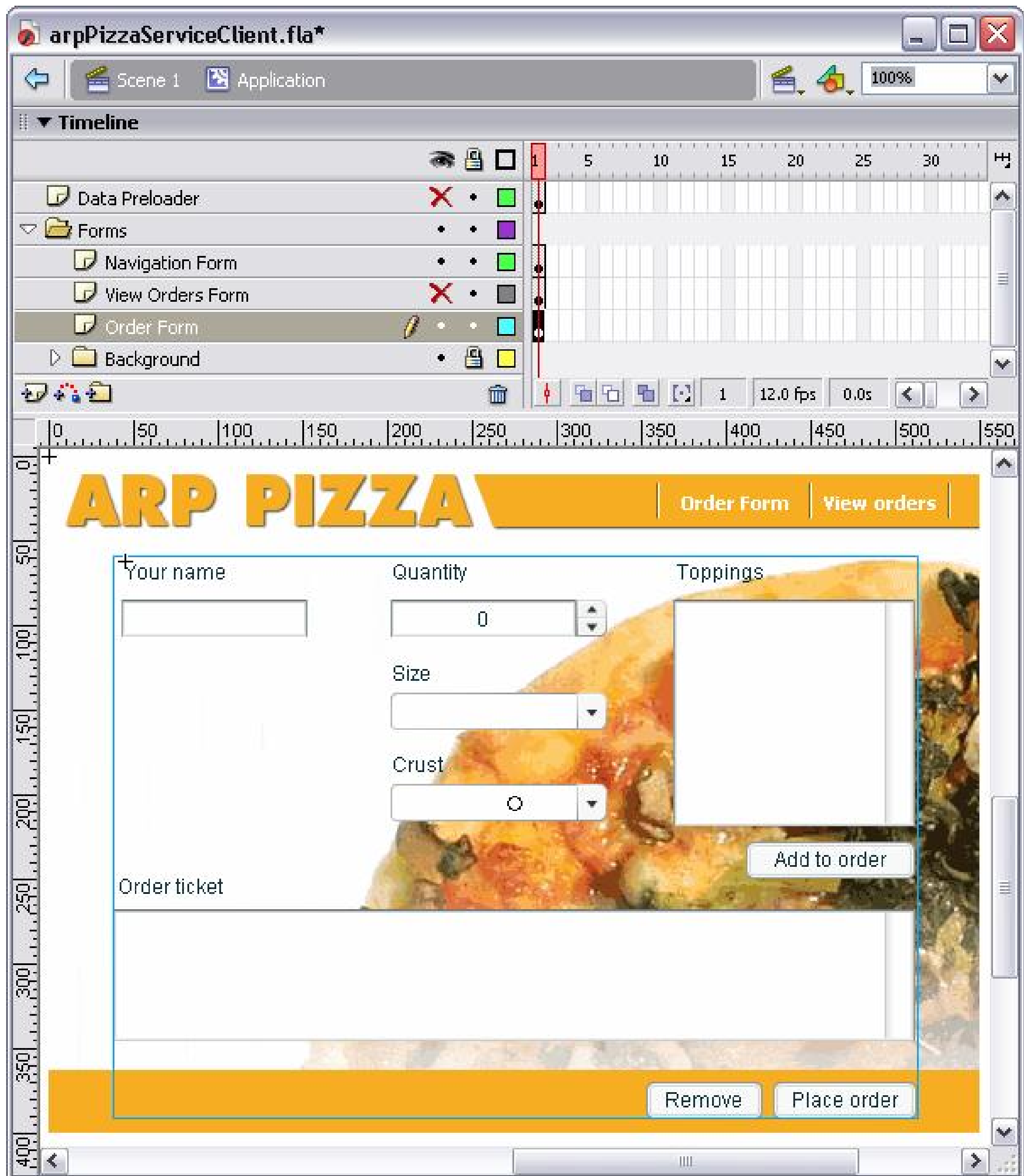
It is beyond the scope of this article to show you how to set up your development environment and install the various servers and the database necessary to build and run the Flash version of ARP Pizza. You can do this on your own, if you want, by downloading ARP (http://osflash.org/ARP) and following the instructions in the comprehensive FlashHelp manual it comes with. The Flex version of the application, which you shall see later in the article, will be released in the next official update to ARP. The latest pre-release version is currently available for download from the ARP Subversion repository at SourceSecure (http://svn.sourcesecure.co.uk/osflash/arp).

Instead of providing a step-by-step tutorial, I will highlight the important architectural elements of the ARP Pizza application, namely its forms-based design and the separation of business and presentation logic and examine how these two factors make it very simple to migrate it from Flash to Flex. I will then take you on a tour of the Flex version, highlighting the changes.

Forms-based design

The ARP Pizza application uses a forms-based design utilizing light-weight, movie clip-based ARP Forms. The ARPForm class extends the core MovieClip class using the EventDispatcher mixin to allow the broadcasting of events.

The ability to dispatch events is central to implementing loose coupling in your applications and makes it possible for you to cleanly separate presentation and business logic. (I could just as easily have used Flash MX 2004 Professional Forms in their place but I didn't need the additional functionality – or weight – in such a simple application.)



[ARP Pizza FLA]

When you open up the ARP Pizza FLA, the first thing you notice is that there is no code anywhere to be found. This is very important. As I mentioned earlier, the FLA is a proprietary, binary format and putting code in it is just asking for trouble by making your applications harder to maintain and scale.

Also apparent is that there is only a single frame in the main timeline and that, in turn, contains a single ARP Form: A special form called the Application form. The Application form is special because it contains, in a hierarchy, all of the other forms that together make up your application's View.

This structure is a natural construct for Flash as it results in child forms getting initialized before parent forms, all the way up the hierarchy until we reach the core Application form. Among other things, this does away with having to create an additional static main() method or other entry point into your application. The onLoad() method of the Application form becomes your entry point.

All of the forms are linked to their respective ActionScript 2 classes through their Linkage IDs. (The ApplicationForm, for example, is linked to the com.ariaware.pizza.view.Application class.)

In the screenshot of the ARP Pizza FLA, you can see what the inside of the Application form looks like in the Flash IDE. It contains a Data Preloader component, which is shown during data exchanges, child forms and some background graphical elements. The Order form is shown as selected.

Componentware

The Order form, in turn, is made up of standard Version 2 Flash components. This is also important. Apart from providing a standard library of common components, the Version 2 components all use the EventDispatcher and are implemented using the Model-View-Controller pattern. This makes them ideally suited for use in an event-based system like ours. Furthermore, Flex components are based on the *same* codebase and differ only slightly from their Flash counterparts. This makes migration easier. In fact, it would be logical to expect future releases of Flash and Flex to bring the components closer together until we have a single component set for both.

Separating Presentation Logic and Business Logic

So as you can see in the screenshot, the FLA in our example contains the application's View which is broken down into a hierarchy of forms, with the main Application form containing the Order form, Navigation form and View Orders form. Each of the forms is linked to its own class. The Application form is special in that, as the root form, it acts as the Controller for the View itself – listening for and responding to view-related events. But what about other events? Events that should somehow trigger business logic to be executed such as when the user presses the Place Order button to order some pizza? In ARP, these are known as System Events and we use an Application Controller to listen for them and execute Commands in response.

Controller, Commands, Value Objects and the Service Locator

When the user presses the Place Order button, the button component broadcasts a "click" event. In response to this, the placeOrder() event handler in the Order Form gets called. This happens because the Order Form registered that method as a listener with the Place Order button when the form first loaded:

```
// Listen for the Place order button click
placeOrderButton.addEventListener ( "click", Delegate.create ( this, placeOrder ) );
```

(The Delegate class is a new addition in the 7.02 updater for Flash and causes the "click" event to be relayed to the placeOrder() method.)

The placeOrder() event handler, in turn, dispatches an event that is unique to the Order Form, called "orderPizza". The method is shown below:

```
function placeOrder ():Void
{
	dispatchEvent ( { type: "orderPizza" } );
}
```

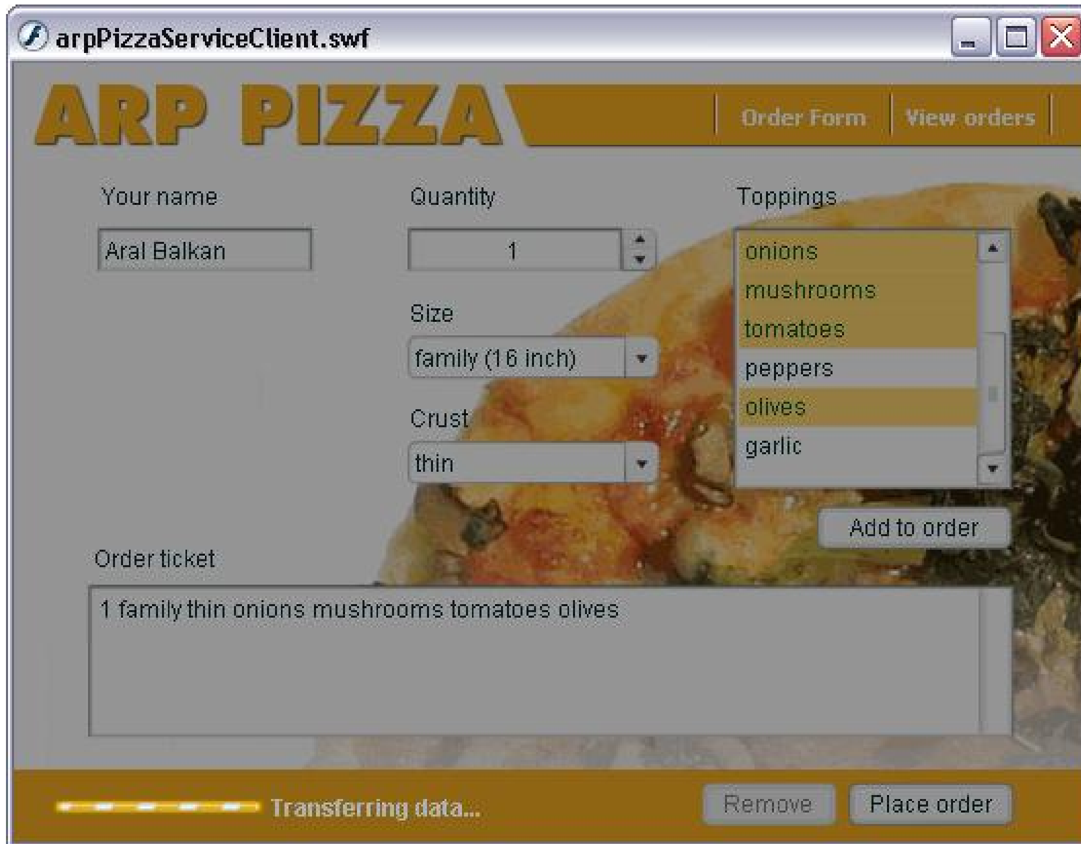
The Application form, in its role as the Controller for the child forms in the View, is listening for the "orderPizza" event, having registered itself as a listener when it first loaded:

```
orderForm.addEventListener ( "orderPizza", this );
```

In the Application form's orderPizza() event handler, there is just one line of code:

```
function orderPizza ()
{
	showDataPreloader();
}
```

This causes the Data Preloader movie clip, which contains a standard indefinite ProgressBar component along with a screen shade/button blocker, to display.



[ARP Pizza application running with the Data Preloader visible]

The screen shade/button blocker prevents users from pressing the Place Order button multiple times – a common usability issue that exists in many HTML-based web applications.

But wait a minute, how does the client notify the server that the user has ordered a pizza? There must be something else listening for the “orderPizza” event on the Order Form and there is: The Application Controller.

The Application Controller listens for System Events on the View and executes Commands in response. Commands can carry out business logic themselves, or defer that task to a dedicated Business Delegate. (In this simple example, I chose to forgo the use of a Business Delegate.)

In ARP, the Application Controller is set up so that it automatically maps System Events to Commands using a well-defined naming convention. Receiving an “orderPizza” event, for example, results in the creation of an instance of the OrderPizzaCommand class and the subsequent invocation of its executeOperation() method. This method is shown below:

```
public function executeOperation ()
```

```
{
	// Get the Order VO
	var orderVO:OrderVO = viewRef.getOrderVO();

	var pizzaService:Service = ServiceLocator.getInstance().getService (
"pizzaService" );

	// We cannot pass the OrderVO directly since the service is hosted remotely and
	// we cannot refactor it to accept a VO.
	var pendingCall:PendingCall = pizzaService.order( orderVO.name,
orderVO.orders );

	pendingCall.responder = new RelayResponder(this, "onResultOperation",
"onStatusOperation");
}
```

Here, you can see that we first ask the form that the event originated on (and which the Controller provided a reference to us with in the viewRef property) to send us the Order Value Object. A Value Object is nothing more than a single, simple object that contains all the data we need to carry out a specific unit of business logic. In this case, it contains all the properties we need to successfully process an order. By passing around a Value Object instance instead of multiple arguments, we make our application easier to maintain. Furthermore, since the server-side implementation will have an identical Value Object, this forms a very strong contract between the tiers. Members of both the client-side and server-side teams can easily check this contract for consistency at any time.

In this example, we are forced to call a remote method using multiple arguments as we are accessing the original back-end provided by the AMFPHP example instead of our own local instance. I decided to implement it in this manner to make it easier for developers to run the sample application upon downloading ARP without having to first install a web server, PHP, MySQL and AMFPHP on their development machines. Ideally, the remote method call would have looked like this:

```
var pendingCall:PendingCall = pizzaService.order ( orderVO );
```

The remoting call is carried out using the new Flash Remoting ActionScript 2 API which utilizes the RelayResponder class to relay result events to the onResultOperation() method and status (error) events to the onStatusOperation() method.

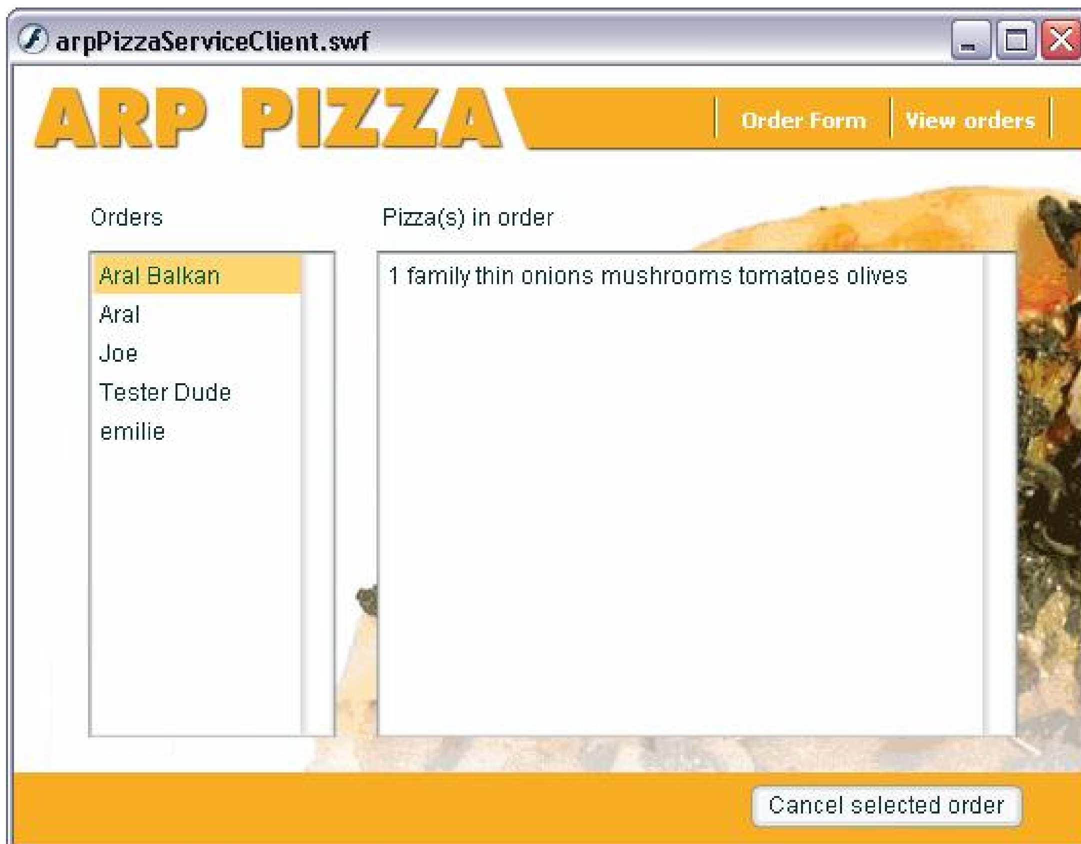
To complete the chain of events sparked off by the user pressing the “Place order” button in the Order Form, take a look at the onResultOperation() method that gets called once the remoting method has returned:

```
function onResultOperation (resultObj:ResultEvent):Void
{
	// Alert the View that the command executed successfully
```

```
        viewRef.orderProcessed();
}
```

The orderProcessed() method is called to inform the form that the order has been processed. The orderProcessed() event handler on the Application form responds by hiding the Data Preloader and making the View Order form active so the user can see that the order is added to their list of orders.

```
function orderProcessed ()
{
        hideDataPreloader();
        viewOrdersFormSelect();
}
```



[ARP Pizza application with the View Orders form showing after a new order is placd]

This quick run-through should be enough to demonstrate the following Five Golden Rules of Flash Application Architecture that are common to any well-architected Flash application:

The Five Golden Rules of Good Flash Application Architecture

1. The FLA is only used to layout the View.

2. The View is comprised of Forms which follow a hierarchical structure with a single root Application form that contains child forms. Whether these are ARPForms or Flash MX 2004 Professional Forms or your own amazing creation doesn't matter as long as they can broadcast events and encapsulate their contents.

3. The forms, in turn, contain well-encapsulated components and the whole application practices a componentware approach to development.

4. All code is kept externally in the form of ActionScript 2.0 classes. The forms in the View are linked to the external ActionScript classes using Linkage IDs.

5. The presentation and business logic are separated using well-known design patterns like the Application Controller, Command, Business Delegate and Service Locator.

Once we have a Flash application that satisfies the above five points, it is simple to migrate it to Flex. In fact, apart from small changes to the View classes to handle component API inconsistencies between Flash and Flex, all we have to do re-create the layout of our forms and components in MXML instead of in the FLA.

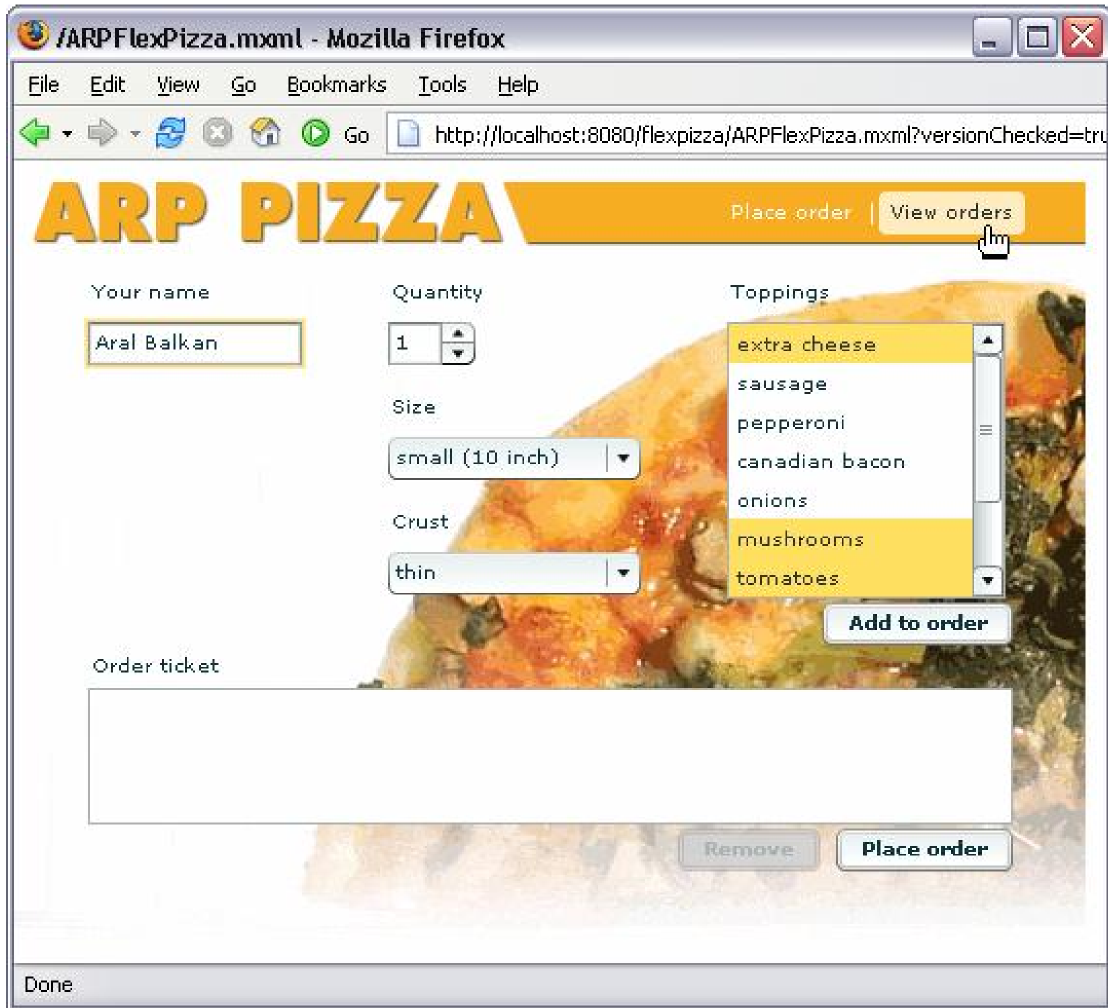
ARP Pizza: Flex Version

In migrating the ARP Pizza Flash application to Flex, I implemented the following changes:

1. Instead of having all the forms in the same MXML file (comparable to having all the forms in the same FLA), I chose to use different MXML files for each form. Flex makes this easy to do and modularizing your application in this way is a best practice (doing the same thing for the Flash version would have complicated the example with extra external form-loading logic that Flex handles automatically.)

2. Instead of the simple custom Navigation form, I used the provided LinkBar component in Flex and included this in the main Application form. (A Flash version of this component does not currently exist.)

3. To create and manage the hierarchical structure of forms, I used the ViewStack component in Flex.



[The Flex version of ARP Pizza running in Mozilla Firefox, with the View Orders link in the LinkBar component highlighted]

The MXML code for the Flex version of the main Application form looks like this:

```
<?xml version="1.0" encoding="utf-8"?>

<pizzaView:Application
        xmlns:pizzaView="com.ariaware.pizza.view.*"
        xmlns:mx="http://www.macromedia.com/2003/mxml"

        backgroundImage="@Embed('Graphics/design.gif')"

        width="550" height="400"
        marginTop="0"
        marginBottom="0"
```

```
        marginLeft="0"
        marginRight="0"
>
        <mx:Spacer height="10"/>
        <mx:LinkBar dataProvider="screens"/>
        <mx:ViewStack id="screens">
    <pizzaView:OrdersForm id="orderForm" label="Orders Form" />
    <pizzaView:ViewOrdersForm id="viewOrdersForm" label="View Orders" />
        </mx:ViewStack>
</pizzaView:Application>
```

Notice how, just as it was with the FLA, there is no code in the MXML file. Putting code in your MXML file can become just as bad a maintenance nightmare in the long run as putting code in your FLA. The only place your code belongs is in your ActionScript 2 classes.

Examining the XML structure, you will see that I first define a custom namespace called pizzaView that points to my view package for this application. This is where my form classes (Application, OrdersForm and ViewOrdersForm) reside.

In my Application class, instead of subclassing an ARPForm or a Flash MX 2004 Professional Form, I subclass the core Flex Application form:

```
class com.ariaware.pizza.view.Application extends mx.core.Application
```

The big difference is that in Flex, the onLoad() method does not get called like it does in Flash. Instead, you have to override the init() method of the mx.containers.Container class (a superclass of mx.core.Application) and listen for the “childrenCreated” event. I’ve found that this is actually a much more precise event than onLoad() in Flash in that it unfalteringly gets called when a component or form’s children have fully loaded and initialized (in Flash we have to sometimes resort to a slew of doLater() tricks to wait a single frame before referencing certain components.)

The init() method that is added to the Application class is shown below:

```
function init ():Void
{
        super.init();

        addEventListener ( "childrenCreated", Delegate.create ( this, onLoad ) );
}
```

As you can see, to minimize code changes, I am relaying the “childrenCreated” event to the existing onLoad() event handler.

Apart from slight changes to accommodate the new LinkBar component, the rest of the class remains the same.

Similarly, the other two forms subclass mx.containers.Form instead of ARPForm and also implement the new init() method.

The View's API, consisting of its public methods and events, remains the same. Most importantly, *I don't have to change a single line of code anywhere else in my application!* Not in my Application Controller, not in any of my Commands and not in my Service Locator. All of my business logic stays the same (as it should, because my business rules have not changed). The only thing that I have to change is my presentation layer (View). Thus, migrating your well-architected Flash application to Flex is merely a matter of recreating your application's View in Flex.

In fact, I found that the most difficult part of the process was trying to get the Flex version to look as similar to the Flash version as possible. By default, Flex offers a wonderful assortment of layout containers that make building fluid, resizable interfaces child's play. However, it takes a bit more discipline if you have to make an interface conform to an exact design with pixel-perfect precision. It is, by no means impossible to do so, however, as the above example demonstrates.

Flash to Flex migration is a breeze!

To recap, if you have followed the Five Golden Rules in creating your Flash application, migrating it to Flex involves only the following three steps:

1. Re-create your forms and lay out your User Interface using MXML.

2. Modify your View classes so that your Application form extends mx.containers.Application and your child forms extend mx.containers.Form and implement an init() method on your forms to listen for the "childrenCreated" event and relay it to your onLoad() method.

3. Implement any changes that may be necessary in your View to accommodate new Flex components you have added that don't have Flash counterparts.

In other words, to migrate your Flash application to Flex, simply recreate your View in Flex because that is all Flex is – a different way of rendering views for Flash-based Rich Internet Applications.

This has been a whirlwind tour of migrating a well-architected Flash application to Flex. I hope it has helped you see that the two technologies are more similar than they are different and that moving from Flash to Flex can be a simple process. It also becomes clear that the most important element in the Flash ecosystem today is ActionScript 2. By externalizing all code in your FLA and MXML files using ActionScript 2 and following

the Five Golden Rules, you will ensure that your Rich Internet Applications are easy to maintain and scale and even migrate from Flash to Flex.

Aral Balkan – Bio

Aral is founder and director of Ariaware, a leading consulting and training firm based in Brighton, UK and specializing in providing usability design and testing, training, consulting and architecture and development for Flash and Flex-based Rich Internet Applications. He is a Macromedia Certified Instructor, author of the Ariaware RIA Platform, runs OSFlash.org and is director of the London Macromedia User Group. You can get information on his other open-source tools, read tutorials and get the latest Flash and Flex news, information and commentary on his blog, FlashAnt (http://flashant.org).